In 2nd International Workshop on Cache Replacement Championship (CRC'17)

Reuse-Aware Management for Last-Level Caches

Priyank Faldu
priyank.faldu@ed.ac.uk

Boris Grot
boris.grot@ed.ac.uk

Institute for Computing Systems Architecture (ICSA)
School of Informatics, University of Edinburgh

ABSTRACT

Variability in generational behavior of cache blocks is a key challenge for cache management policies that aim to identify dead blocks as early and as accurately as possible to maximize cache efficiency. Existing management policies are limited by the metrics they use to identify dead blocks, leading to low coverage and/or low accuracy in the face of variability. In response, we introduce a new metric – *live distance* – that uses the stack distance to learn the temporal reuse characteristics of cache blocks. We further introduce Leeway, a new dead block predictor that leverages live distance to enable dead block predictions that are robust to variation in generational behavior. Based on the reuse characteristics of application's cache blocks, Leeway classifies its behavior as *streaming* or *reuse* and dynamically selects an appropriate cache management policy.

1. INTRODUCTION

Dead Block Predictors (DBPs) have been shown to be effective in improving cache performance through better utilization of existing capacity [1,2,3,4,5,6]. These schemes all rely on some metric of temporal reuse to make their decisions regarding the end of a given block's useful life. Previous work has suggested hit count [1], last-touch PC [2], and number of references to the block's set since the last reference [7], among others, as metrics for determining whether the block is dead at a given point in time. By identifying and evicting dead blocks in a timely and accurate manner, these schemes allow other blocks (that have not exhausted their useful life) to persist in the cache and see further hits.

The task of a DBP is complicated by the fact that applications exhibit *variability* in the re-reference patterns of cache blocks touched by them. The sources of variability are numerous, stemming from microarchitectural noise (e.g., speculation), control-flow variability, cache pressure from other threads, etc. The variability manifests itself as an inconsistent behavior of the individual cache blocks from one cache lifetime, or generation, to the next. This inconsistency challenges DBPs in reliably identifying the end of a block's useful lifetime, thus resulting in lower prediction accuracy, coverage, or both.

Our thesis is that DBPs require metrics and policies that can tolerate inconsistencies. To that end, we propose *live distance* – a metric of temporal reuse based on stack distance. For a given cache block, live distance is the largest observed stack distance in a generation (from allocation to eviction). Live distance demarcates the range of the block's temporal reuse within the LRU stack. When the block's position within an LRU stack exceeds its known live distance, the block is unlikely to be referenced and can be predicted dead. Thus, live distance provides an efficient way to represent a block's range of temporal use.

To obtain stack distance values, we exploit the fact that LRU-based policies implicitly track stack distances of cache-resident blocks. In true LRU, when a block hits, its current LRU stack position corresponds to its stack distance. For policies that deviate from true LRU, such as multi-bit NRU (see Sec. 2 for details), a block's stack position upon a hit only approximates the true stack distance. Nevertheless, it provides an efficient heuristic to approximate stack distance and, correspondingly, live distance.

We introduce Leeway, a new DBP that uses live distance as a metric for prediction. Leeway uses code-data correlation to associate live distance for a group of blocks with a PC that brings the block into the cache. While live distance as a metric provides a high degree of resilience to variability, the per-PC live distance values themselves may fluctuate across generations. To correctly train live distance values in the face of fluctuation, we observe that individual applications' cache behavior tends to fall in one of two categories: *streaming* (most allocated blocks see no hits) and *reuse* (most allocated blocks see one or more hits). Based on this simple insight, we design a pair of corresponding policies that steer updates in live distance values either toward zero (for bypassing) or toward the maximum recently-observed value (to maximize reuse). For each application, Leeway picks the best policy dynamically based on the observed cache reuse behavior.

2. LEEWAY BASICS

In this paper, we focus on Leeway with a low-cost multi-bit *Not Recently Used (NRU)* family of policies. Multi-bit NRU uses two or more bits per cache block to indicate a partial relative order of LRU stack positions. For instance, a 2-bit NRU policy keeps blocks in a set in one of four equivalence classes as a function of their relative stack positions, with class 0 for MRU blocks and class 3 for LRU ones. During victim selection, a block in class 3 is evicted (ties are broken through random selection). If no block is found in class 3, every block is moved to the next class and the process is repeated.

For the competition, we implement Leeway on such a hardware-friendly 2-bit NRU where block's NRU value is

Source code: https://github.com/faldupriyank/leeway used to approximate its stack distance, and in turn, live distance. However, in the discussion below, we refer to NRU values as its LRU stack position to keep the discussion generic.

Leeway records the maximum observed hit position (i.e., live distance) during a block's residency in the cache. At eviction time, the live distance is recorded in a separate structure, *Live Distance Predictor Table (LDPT)*, for subsequent recall when the block is allocated again. Leeway uses the live distance learned in the block's previous generations to infer when the block may have exceeded its useful lifetime and predicts it dead. To avoid the prohibitive storage costs of tracking individual cache blocks in the LDPT, Leeway exploits code-data correlation and associates all cache blocks allocated by a given PC with one LDPT block.

The functionality of Leeway can be divided into three categories - *Learning, Prediction* and *Update*. Learning is a continuous process for cache-resident blocks that involves checking a block's position in the LRU stack upon each hit and, if the current position exceeds the past maximum, updating the live distance. Prediction is triggered during victim selection on a miss to a set. Any block that has moved past its predicted live distance in an LRU stack is predicted dead. Update occurs upon a block's eviction from the cache, propagating the new live distance to the LDPT. To effectively handle variability in live distance across generations of a given block and across blocks tracked by a given LDPT entry, the update process is conditional as explained in Section 3.

Leeway implements set-sampling, similar to [8], to learn the blocks' live distances by observing their behavior in a small number of sample sets. The reason for sampling is two-fold: 1) it helps filter out some of the noise in observed live distances; 2) it significantly reduces Leeway's storage requirement as only blocks belonging to the sample sets need to be augmented with storage and logic needed for learning.

3. REUSE AWARE POLICIES

Due to numerous reasons, a block's observed reuse behavior may fluctuate in time even if its fundamental reuse characteristics are not changing. While the live distance metric provides a degree of protection from intra-generation noise, Leeway must contend with inevitable fluctuation in live distance across generations and across different blocks allocated by the same PC. In particular, it must separate unrepresentative live distance values from actual shifts in reuse behavior. This observation points to the need for an intelligent update policy for Leeway's live distance values.

To design a variability-tolerant update policy, we study both SPEC and scale-out server workloads (CloudSuite) to understand their reuse behavior. Our workload analysis reveals that applications tend to fall in one of two categories in terms of their reuse behavior affecting LLC management.

The first category is dominated by streaming accesses that do not observe any LLC hits and should be bypassed. In many cases, the blocks allocated by certain streaming PCs will occasionally observe one or more hits. Moreover, such behavior sometimes occurs in clusters, forcing a shift in cache management policy from bypassing to keeping blocks on chip. Such a shift is generally undesirable, as the behavior tends to quickly revert back to streaming. A multi-bit hysteresis threshold may be effective in delaying a shift in policy; however, the high threshold is counter-productive when the behavior reverts back to streaming as it will lead to blocks being allocated in LLC rather than be bypassed.

The second category of applications is dominated by blocks that do see reuse prior to being evicted from the LLC. However, we observe considerable variation in live distance for many PCs that allocate blocks exhibiting reuse. This observation is consistent with prior work that observed that blocks with reuse are more prone to variation in inter-generational behavior than streaming blocks, thus posing a challenge for dead-block predictors [9]. Given the uncertainty in reuse behavior, such blocks should be kept longer to maximize opportunity for reuse.

Based on the above observations, we propose two separate policies for each type of behavior to maximize bypass opportunities for streaming workloads and reuse for workloads that exhibit it.

Bypass-Oriented: This policy seeks to maximize opportunities for bypass by being slow to increase the live distance and fast in dropping it back towards 0. An incoming block with a predicted live distance of -1 is bypassed, unless it maps to a sampler set (see Sec. 4.1.1 for details).

Reuse-Oriented: To maximize reuse opportunities for allocated blocks when there is fluctuation in live distance values, this policy is quick to increase the live distance and slow to decrease it. Since Leeway does not evict blocks that have not reached their live distance value in the LRU or multi-bit NRU stack, a larger live distance enables a longer temporal window for uncovering reuse.

The two policies call for diametrically opposite behavior: whereas the Bypass Oriented policy is slow to increase the live distance values in LDPT but fast to decrease them, the Reuse Oriented policy is fast to increase live distance values but slow to decrease them. To satisfy the demand for separate policies in increasing and decreasing live distance in the LDPT, Leeway deploys two *Variation Tolerance Thresholds (VTTs)* that control the rate at which live distance values are adjusted based on workload behavior and the direction of change in live distance.

In order to choose the preferred policy for a running application, Leeway leverages Set Dueling [8] and implements both policies (Bypass- and Reuse- Oriented) simultaneously on separate sampler sets. The rest of the cache follows the policy that minimizes the misses.

4. MICROARCHITECTURE DESIGN

4.1 Physical Fields and Structures

Fig. 1 summarizes key elements of the design.

LDPT: LDPT is implemented as a set-associative structure with small associativity. Each LDPT block is augmented with *LRU-stack*, *tag* and a *valid* bit to implement it with LRU replacement policy. LDPT block further contains two LDPT entries - one for each Bypass- and Reuse- Oriented Policy. Each LDPT entry contains a *stable-live-distance* field that indicates the current live distance based on most recent history. Updates to stable-live-distance are controlled by VTTs and two additional LDPT fields: 1) *variance-count* is a counter for tracking the number of consecutively evicted

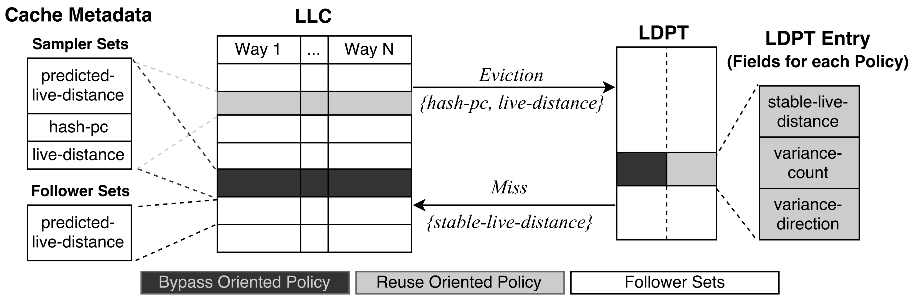

Figure 1: Schematic of Leeway for LLC

cache lines whose live distance differs from the stored value, and 2) *variance-direction* is a bit indicating the direction of the difference. Once the count matches the value of a VTT for a given direction, the value of *stable-live-distance* is updated. To avoid additional storage for transient live distance values, the new *stable-live-distance* value is taken from the evicted block that triggers the update.

VTTs: To enable Bypass- and Reuse- Oriented policies, Leeway uses a pair of Variation Tolerance Thresholds that control the rate at which *stable-live-distance* values are updated. Empirically, we find that a 3-bit VTT is sufficient, and use the maximum value for the slow update (i.e., requiring 7 consecutive evictions with a live distance different, and in the same direction, from the *stable-live-distance*) and a value of 1 for the aggressive threshold. Thus, the two valid VTT configurations are either {7,1} (for the Bypass-Oriented policy, with a slow increase and fast decrease) and {1,7} (for the Reuse-Oriented policy with a fast increase and slow decrease).

LLC: Leeway requires all LLC blocks to carry a field, *predicted-live-distance*, which is read from the LDPT at block allocation time and is subsequently used for dead block prediction. Sampler sets carry two additional fields: *live-distance* & *hash-pc*. These are used for learning, allowing evicted blocks to access the LDPT and, if necessary, update its fields as explained above.

4.1.1 Leeway in Action

Cache Miss: On an LLC miss, the LDPT is accessed using a hash of the miss PC to recall the *stable-live-distance*, which is then transferred to the incoming block's *predicted-live-distance* field. If *stable-live-distance* is -1 (i.e., no hits), the block is expected to have no reuse and is bypassed to higher level cache. Because bypassed blocks have no opportunity to retrain, Leeway inserts them into the sampler sets with a small probability to enhance learning.

Cache Hit (Learning): On a hit to a sampler set, the block's live-distance is updated if its stack position is greater than the value of the *live-distance* field. No action for all other cases.

Eviction (Prediction and Update): To find victim, Leeway searches for a dead block by comparing each block's LRU or NRU position to its *predicted-live-distance* field. If more than one dead block is found, a victim is picked at random. If no block is predicted dead, the LRU block is evicted. If the evicted block resides in the sampler set, its *live-distance* and *hash-pc* is forwarded to the LDPT for a potential update.

4.1.2 Mechanism for Policy Selection

To dynamically choose between Bypass- and Reuse- Oriented policies, Leeway relies on set dueling [8]. Thus, two separate groups of sampler sets are used, with each group implementing one of the two policies. Each group of sets always access their dedicated LDPT entry based on a static mapping, the rest of the sets read the *stable-live-distance* from the winning policy.

To determine the winning policy, Leeway maintains two saturating miss counters, one for each policy. The counters are incremented on a miss to a sampler set of a respective policy. Periodically, the miss counters are sampled and the winning policy is selected based on the counter with the lowest value.

Often, the winning policy remains the same throughout the application's execution. In some cases, however, the winning policy may change due to changes in the application's phase or its co-runner(s). In theory, a policy change requires reloading *predicted-live-distance* for all cache blocks using the *stable-live-distance* of the new winning policy in LDPT. In practice, we find that policy change is infrequent, indicating that the simplest way to deal with it is to leave existing blocks untouched, potentially incurring a handful of poor decisions but minimizing microarchitectural complexity.

4.2 Leeway for Multicore

Leeway can be naturally extended to multicore deployments. The only notable difference is in determining the winning policy for each individual core. When extended to multicore, the sampler sets for a given core would be shared with other cores that will use them as followers of their respective (and potentially different) policies. As the choice of a policy used by each core affects other cores, Leeway adopts following strategy.

Leeway maintains *N Miss* and *Access* counters for each core and each policy where *N* is the total number of cores. In a sampler set for a given policy, it tracks misses and accesses by different cores and increments counters appropriately. In a given interval, Miss and Access counters are only incremented till Access counter saturates. This allows comparing misses among applications with different memory intensity fairly. At the end of an interval, the policy is chosen that minimizes total misses.

5. HARDWARE COST AND COMPLEXITY

Cost: We analyze storage requirements for a 16-way 2MB LLC with 64B blocks. In theory, LDPT can be directly indexed by *hash-pc*. However, for applications like CloudSuite with large code-footprint, it may require a large table not to be affected by destructive aliasing. To overcome this, LDPT is designed as a set-associative structure. We find that 512 sets and 4 ways per core is sufficient and is not affected by

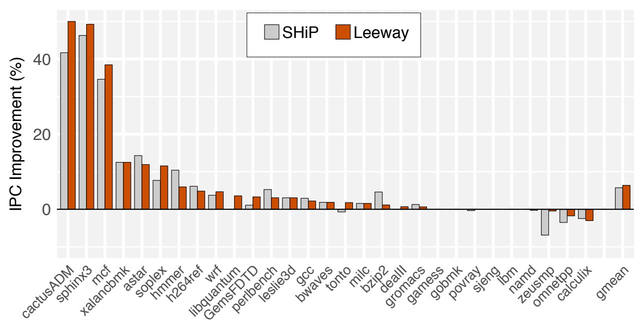

Figure 2: IPC Improvement for SHiP and Leeway when compared to LRU for SPEC applications

destructive aliasing. Number of sets are scaled with number of cores, keeping associativity the same. Replacements in the LDPT are handled via LRU.

Each LDPT block requires a 13-bits for *tag* (for 22-bit hash-pc and 512 sets), 2-bits for *LRU-stack* and 1 *valid bit* of meta data. Every LDPT block stores two LDPT entries - one for each Leeway policy. Each LDPT entry consists of 6 bits: 2 for *stable-live-distance*, 3 for *variance-count* and 1 for *variance-direction*. The resulting cost of each LDPT block is 28 bits, translating to total cost of 7KB for the entire LDPT.

We use a 64-set sampler per policy. Each block in the sampler carries a 2-bit *live-distance* and 22-bit *hash-pc* fields and 1 *policy-selection* bit (not shown in the figure for brevity) requiring 6.25KB of storage in total. All cache blocks, including the sampler, include a 2-bit *predicted-live-distance* and 2-bit *NRU-bits*, totaling 22.25KB of cache storage. The total storage overhead of Leeway is thus 29.25 KB, leaving ample room for accommodating various registers used by the Leeway implementation for a 32KB of storage budget. Table 1 lists the storage requirement of different structures and their fields for 1- and 4-core configurations.

Complexity: Operations performed by Leeway at various stages are limited to simple addition, comparisons and shifts, which are quite hardware friendly. Additionally, Leeway embeds the metadata necessary for the prediction (i.e., *live distance*) with the cache blocks. As a result, LLC hits and replacement decisions never access remote metadata. The only time Leeway accesses its prediction table (LDPT) is upon cache misses, when *stable-live-distance* is read and possibly updated. These accesses are entirely off the critical path, since they do not involve state updates to a live cache block.

In contrast, state-of-the-art DBPs, such as [2, 4, 5, 6], use a PC-indexed prediction table that is probed on every LLC access (including hits) to inform the block's eviction priority. For example, Hawkeye [5] incurs ∼2.3x more accesses to its prediction table when compared to Leeway (SPEC average). Such frequent accesses to the prediction table are particularly undesirable in a modern multicore CPU with a NUCA LLC, as each LLC hit requires state-of-the-art DBPs to access the PC-indexed prediction table located elsewhere on a chip, incurring latency, energy, and traffic overheads due to the need to traverse the on-chip network.

6. RESULTS

We evaluate SHiP [3] as the state-of-the-art technique with complexity comparable to Leeway as both SHiP and Leeway access their predictor tables only on misses. Figure 2 shows the percentage IPC improvement over baseline LRU for all SPEC applications. Overall, Leeway provides geomean improvement of 6.4% vs 5.7% in SHiP. While Leeway outperforms SHiP on 20 out of 29 applications, it does not slow down any application by more than 3%.

Structure	Fields	1-core	4-cores
LDPT Entry	stable-live-distance (a)	2	2
	variance-count (b)	3	3
	variance-direction (c)	1	1
	Total size (d = a+b+c)	6	6
LDPT Block	LRU bits (e)	2	2
	Tag bits (f)	13	13
	Valid bit (g)	1	1
	2 LDPT entries (h = 2*d)	12	12
	Total size (i = e+f+g+h)	28	28
LDPT	Sets (j)	512	2048
	Ways (k)	4	4
	Number of Blocks (l = j*k)	2048	8192
	Total size (m = l * i)	57344	229376
Sampler Block	NRU bits (n)	2	2
	predicted-live-distance (o)	2	2
	live-distance (p)	2	2
	hash-pc (q)	22	22
	policy-type (r)	1	1
	Total size (s = n+o+p+q+r)	29	29
Follower Block	NRU bits (t)	2	2
	predicted-live-distance (u)	2	2
	Total size (v = t+u)	4	4
Cache	Ways (w)	16	16
	Number of Sampler Sets (x)	128	512
	Number of Follower Sets (y)	1920	7680
	Size of Sampler Sets (z = x*w*s)	59392	237568
	Size of Follower Sets (A = y*w*v)	122880	491520
	Total size (B = z+A)	182272	729088
Total storage (bits)	Total size (C= B+m)	239616	958464
Total storage (KB)	Total size (C= B+m)	29.25KB	117KB

Table 1: Leeway storage for 1-core and 4-core configurations

7. REFERENCES

[1] M. Kharbutli and Y. Solihin, "Counter-Based Cache Replacement and Bypassing Algorithms," *IEEE Transactions on Computers*, vol. 57, no. 4, pp. 433–447, April 2008.

[2] S. M. Khan, Y. Tian, and D. A. Jiménez, "Sampling Dead Block Prediction for Last-Level Caches," in *Proceedings of the International Symposium on Microarchitecture*, December 2010, pp. 175–186.

[3] C.-J. Wu, A. Jaleel, W. Hasenplaugh, M. Martonosi, S. C. Steely, Jr., and J. Emer, "SHiP: Signature-based Hit Predictor for High Performance Caching," in *Proceedings of the International Symposium on Microarchitecture*, December 2011, pp. 430–441.

[4] E. Teran, Y. Tian, Z. Wang, and D. A. Jiménez, "Minimal Disturbance Placement and Promotion," in *Proceedings of the International Symposium on High Performance Computer Architecture*, March 2016, pp. 201–211.

[5] A. Jain and C. Lin, "Back to the Future: Leveraging Belady's Algorithm for Improved Cache Replacement," in *Proceedings of the International Symposium on Computer Architecture*, June 2016, pp. 78–89.

[6] E. Teran, Z. Wang, and D. A. Jiménez, "Perceptron Learning for Reuse Prediction," in *Proceedings of the International Symposium on Microarchitecture*, October 2016, pp. 1–12.

[7] N. Duong, D. Zhao, T. Kim, R. Cammarota, M. Valero, and A. V. Veidenbaum, "Improving Cache Management Policies Using Dynamic Reuse Distances," in *Proceedings of the International Symposium on Microarchitecture*, December 2012, pp. 389–400.

[8] M. K. Qureshi, D. N. Lynch, O. Mutlu, and Y. N. Patt, "A Case for MLP-Aware Cache Replacement," in *Proceedings of the International Symposium on Computer Architecture*, June 2006, pp. 167–178.

[9] P. Faldu and B. Grot, "LLC Dead Block Prediction Considered Not Useful," in *Workshop on Duplicating, Deconstructing and Debunking*, June 2016.